Design and Implementation of Real-Time Transactional Memory

Schoeberl, Martin; Hilber, Peter

Published in:
Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL 2010)

Link to article, DOI:
10.1109/FPL.2010.64

Publication date:
2010

Document Version
Early version, also known as pre-print

Design and Implementation of Real-Time Transactional Memory

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

Peter Hilber
Institute of Computer Engineering
Vienna University of Technology, Austria
peter.hilber@student.tuwien.ac.at

***Abstract*—Transactional memory is a promising, optimistic synchronization mechanism for chip-multiprocessor systems. The simplicity of atomic sections, instead of using explicit locks, is also appealing for real-time systems. In this paper an implementation of real-time transactional memory (RTTM) in the context of a real-time Java chip-multiprocessor (CMP) is presented. To provide a predictable and analyzable solution of transactional memory, the transaction buffer is organized fully associative. Evaluation in an FPGA shows that an associativity of up to 64-way is possible without degrading the overall system performance. The paper presents synthesis results for different RTTM configurations and different number of processor cores in the CMP system. A CMP system with up to 8 processor cores with RTTM support is feasible in an Altera Cyclone-II FPGA.**

I. Introduction

Transactional memory (TM) provides a single, simple mechanism for mutual exclusion: atomic sections. The implementation is in charge to ensure that either the whole transaction is committed to main memory or retried. TM can be implemented in software or in hardware. In this paper we describe the implementation of hardware transactional memory (HTM).

In [15] we have presented the idea of real-time transactional memory (RTTM) and showed that the number of retries is bounded. It has also been showed how program analysis can be used to reduce the number of possible conflicting transaction. In this paper we present the concrete design and implementation of RTTM in a Java chip-multiprocessor (CMP) system. As processor we use JOP, where worst-case execution time (WCET) analysis is supported [14]. The design is evaluated in an Altera Cyclone-II FPGA. The resulting resource consumption for small transaction buffers is reasonable and the maximum frequency in the same range as a CMP without RTTM support.

RTTM brings the benefits of transactional memories into the real-time systems world. It simplifies the programming model with the introduction of atomic regions instead of locks. Error prone fine grain locking or lock-free algorithms can be avoided. The contributions of the paper are: (a) the design of a time-predictable hardware transactional memory; and (b) evaluation of the design in an implementation in an FPGA.

The paper is organized as follows. Section II presents related work. The design of RTTM is presented in Section III, followed by a description of the representation of transactions in Java in Section IV. The concrete implementation is presented in Section V. The paper is concluded by Section VI.

II. Related Work

Knight was the first to propose hardware support for transactions [7]. In his paper he considers support for mostly functional languages. The key elements are two fully associative caches: the *depends cache* implements the dependency list (besides acting as normal data read cache) and the *confirm cache* acts as local cache for uncommitted writes (side effects in his terminology).

The term *transactional memory* was coined by Herlihy and Moss [5]. They realized that only a minor modification of the available cache coherence protocols is needed to implement transactional memory.

Shavit and Touitou present *software transactional memory* (STM) [17]. The proposed STM provides static transactions. That means the data set has to be known in advance. Saha et. al. propose an ISA extension to provide architectural support for STM [11]. The idea is based on additional mark bits for parts of a cache line (e.g., for 16 byte blocks of a 64 byte cache line). Language support for transactions [4] in Java reinvestigates Hoare's conditional critical regions. To distinguish between normal field access and field access under a transaction a second method table holds references to a transactional version of methods. In [6] a practical implementation as a library (DSTM2) for standard Java is presented that requires only compare-and-swap (CAS) instructions.

The Transactional memory Coherence and Consistency (TCC) model is proposed in [3]. TCC combines the simpler hardware for message passing with the simpler shared memory programming model. The standard cache coherence protocol with the latency issue on each load and store instruction is substituted by the TCC hardware. The TCC hardware broadcasts all writes from each transaction in a single packet. Automatic rollback resolves any conflicts. TCC differs from other approaches as *all* instructions are part of a transaction. The code is just split into transactions, which

can be done manually or automatically by the hardware. Language extensions for loop and fork based parallelization for TCC are presented in [2]. The paper also contains detailed simulation results of speedup and write set sizes. The speedup is reported in the range of 4.5 to 7.8 for a 8 processor CMP configuration. For most applications a write buffer of 1 KB is sufficient. We assume that applications in the real-time domain will need even less on-chip memory. Also a finer granularity of the write buffer (single words instead of 64 Byte cache lines) will reduce the needed size.

A first prototype of TCC in FPGAs [18] implements a 8 processor configuration with PowerPC cores and a custom data cache for the transaction buffer. The system consists of 4 FPGAs for the TCC (2 PowerPCs per FPGA) and one control FPGA that runs Linux. Although the PowerPC can be clocked up to 300 MHz, the TCC system is clocked with 100 MHz. The paper also reports issues with the implementation of the custom data cache in current FPGA technology. The 4-way set associative TCC cache has an access time of 13 clock cycles and clearing the cache state bits at the end of a transaction takes 257 clock cycles. An on-chip block RAM is used for the register checkpoint.

In contrast to the TCC implementation, we implemented the transaction buffer with a high associativity with single cycle access. As our transaction buffer is organized in FIFO order, the transaction buffer can be cleared within a single cycle by resetting a single pointer.

RTTM shares many ideas with TCC. We also perform late conflict detection at commit and grab the commit token early on a buffer overflow. However, the design of the transaction buffer in our approach is different: TCC uses a standard cache organization for the transaction buffer, while we optimize our design for time predictability and not for average case throughput. Furthermore, TCC uses transactions for *all* memory operations. The resulting high variability of memory access times is hard to include in the WCET analysis.[1] We use the TM only for short atomic code sections and perform non transactional loads and stores via a time-predictable memory arbiter [9]. The TCC design consumes about 8000 LCs per processor core (not including the processor). In our implementation the buffer for 32 words consumes moderate 1600 LCs.

Preemptible atomic regions (PAR) [8] was the first proposal of TM for real-time systems. A PAR is aborted when a higher priority task becomes ready and preempts the lower priority task – independent whether the high priority task executes an atomic region or not. The effects of a PAR are undone at the interrupt. The concept of PAR is only valid on uniprocessor systems.

[1]The bounds become impractical high: for n processors the WCET of a single load or store is $n-1$ times the longest commit time.

III. Real-Time Transactional Memory Design

The mechanics of a transaction in RTTM is as follows: At transaction start the state of the processor is saved. During the transaction all memory writes – the update of the global state – go into a core local buffer. On a commit the local state is atomically written to the main memory. If a conflict between a committing transaction and another transaction occurs, the other transaction is restarted. On a restart the local memory content is discarded and the processor reset to the saved state.

The main design goals for the RTTM are: (a) simple programming model and (b) analyzable timing properties. Therefore, all design and architecture decisions are driven by their impact on the WCET. In contrast to other TM proposal RTTM does not aim for a high average case throughput, but for a time-predictable TM with a low WCET. RTTM is intended to support small atomic sections with a few read and write operations. Therefore, it is more an extension of the CAS instruction to simplify the implementation of non-blocking algorithms.

A. RTTM Analysis

For real-time systems the execution time of all operations must be bounded and the bounds need to be known for WCET analysis. In case of transactions, the number of retries needs to be bounded. In the general case, e.g., performing transactions in tight loops, the retry count of a transaction is unbounded. The common solution is to use a transaction manager and an exponential back-off, which works in practice.

For real-time systems stronger bounds on the retries are needed. In [15] we have shown that if the transactions of each thread are displaced at least the conflict resolution time t_r, the maximum number of retries is $n-1$ for n conflicting transactions. As real-time applications are usually organized as periodic tasks, the displacement between transactions is automatically given.[2]

B. Transaction Context

A transaction needs to be restartable with exactly the same context as initially started. The context of a RISC processor is the program counter and all registers that need to be saved and restored on a retry. Local variables and method arguments, which are mapped to memory locations on the thread's stack, can be treated as normal memory locations. They are part of the transaction.

However, in Java method arguments and locals are guaranteed thread local. Therefore, they can be excluded from the transaction. In that case, they become part of the processor context that needs to be saved on transaction start and restored on transaction abort. In Java a method has no access

[2]If tasks that contain more than one transaction per period, the retry count can still be bounded. For details see [15].

to stack allocated data from an outer method. Therefore, this additional context is usually minimal. A simple solution is to add additional local variables to a method and copy the arguments and locals that are visible at transaction start into the additional locals.

C. Transaction Buffer

During a transaction all memory writes go into the core local transaction buffer. Read addresses are marked in a read set – a simplification that uses only tag memories for the read set. Read data can also be cached, but caching is not essential for the correct operation of the RTTM.

Common proposals of hardware transactional memory reuse the core local cache for the transaction buffer. The conflict is detected at cache line granularity. This can result in false positives for conflicts. In the general case the false positives will just degrade the performance. However, for a time-predictable design we have to avoid false conflicts. Therefore, the transaction buffer of RTTM has a granularity of single memory words.

Another issue with standard caches is that two words in memory can map to the same cache line. In that case a backup solution, e.g., early commit, has to be performed. However, the addresses of data in the heap are only known at runtime. Therefore, it is practically impossible to predict such a cache line conflict. In the RTTM design this conflict is avoided by a fully associative buffer. If a transaction accesses less words than the associativity of the buffer, the transaction will not overflow the buffer. Fully associative buffers are expensive and therefore small (e.g., 32 or 64 words). Real-time programs shall be written with this limitation in mind. RTTM can serve as a multi-word CAS to simplify non-blocking algorithms.

D. Transaction Commit

On a commit the content of the transaction buffer is written to the shared memory. During the write burst of the commit all other cores listen to the write addresses and compare those with their own read set. If one of the write addresses matches a read address the transaction is aborted. The atomicity of the commit itself is enforced by a single global lock – the commit token.

If the transaction buffer or the tag memory for the read set overflows, the backup solution is to grab the commit token and finish the transaction while holding the token. If several transactions overflow this mechanism effectively serializes the overflowing transactions – independent if they would conflict or not. The same mechanism can be used to protect I/O operations, which usually cannot be rolled back. On an I/O operation within a transaction the core also grabs the commit token.

E. Conflict Detection

Conflict detection can be performed early, when the first conflict really happens, or late on commit. From the analysis point of view both approaches lead to the same WCET. Early conflict detection is an average-case optimization.

```
void foo(int a, int b) {

        int _a = a;                 // save arguments
        int _b = b;

        while (true) {
                try {
                        RTTM.start();
                        // atomic section
                        RTTM.end();
                } catch (RttmAbort e) {
                        a = _a;
                        b = _b;
                        continue;
                }
                break;
        }
}
```

Figure 1. Code transformation of an atomic method. Method arguments are saved in additional local variables.

Early conflict detection is very expensive in hardware as the buffer local write traffic has to be observed by all other cores. That means that each core has to listen to the other $n-1$ cores. This is the same effort that is needed for a cache coherence protocol. Therefore we propose to use late conflict detection during the commit phase. When one transaction commits its write buffer to the shared memory, all other transaction units just need to listen to the write burst of a single core.

F. Conflict Communication

When a conflict is detected, the corresponding thread can be notified to abort the transaction early or late. Early notification can be represented by a thrown exception. Late notification just marks the transaction for an abort and the abort can be communicated at the end of the transaction. Again from a real-time perspective the worst-case behavior is the same.

The RTTM module communicates a conflict via an exception signal to the pipeline. The mechanism is already in place for exceptions such a null pointer access or division by zero. This hardware exception is translated by the JVM to a standard Java exception, which can be handeled by normal Java try/catch blocks.

IV. Atomic Sections in Java

Transactions are marked atomic by the Java annotation @atomic. As annotations are not possible for arbitrary code blocks, the granularity of a transaction is a method. This method invocation introduces some overhead, but also reduces the context that needs to be saved at transaction start. Only the arguments need to be saved, no local variables, and no possible operands on the JVM stack.

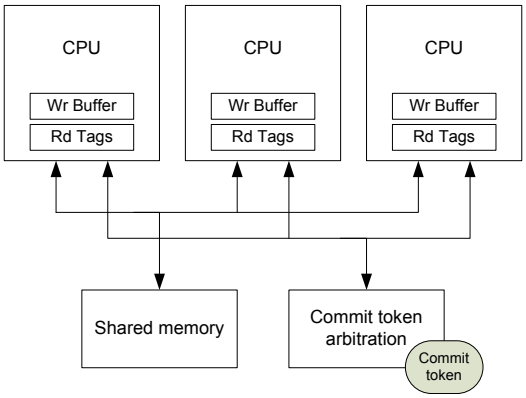

Figure 2. Overview of the CMP system with RTTM

A Java program containing atomic sections is compiled with a standard Java compiler. The transformation of the method is performed by manipulation of the bytecode. We have implemented the transformation within the JOP application builder JOPizer, which itself is based on the bytecode engineering library BCEL [1].

Figure 1 shows a simplified version of the transformed code in standard Java notation. First the method arguments are saved into additional local variables for an eventual restore on a transaction retry. The retry loop is the while loop. The transaction is started with RTTM.start() and the commit is tried with RTTM.end(). If the transaction is aborted, either within the code of the atomic section or the RTTM.end() method, a RttmAbort exception is thrown. The exception is caught, the arguments are restored, and the transaction is restarted.

Details on handling of other runtime exceptions are omitted from the figure. It has to be noted that a transaction can see inconsistent data before it will be aborted. In that case, other runtime exceptions (e.g., NullPointerException) can be thrown, which is also handled by the generated code.

V. Implementation

We have implemented RTTM in the context of the Java processor JOP [12], [14] in an Altera Cyclone EP2C70 on the Altera DE2-70 evaluation board. The main limiting factor is the tag memory for the transaction read and write set. As write locations are often also in the read set, we use a unified tag memory. The tag memory is extended with a read and a write bit. On a commit only the entries where the write bit is set are written back to the memory. The read bit is needed for conflict detection when a transaction on a different core commits.

Besides the core individual transaction buffer, a transaction manager handles the arbitration for the commit token. The interface of the cores to the transaction manager is via a memory mapped device, which is connected with JOP's system bus SimpCon [13].

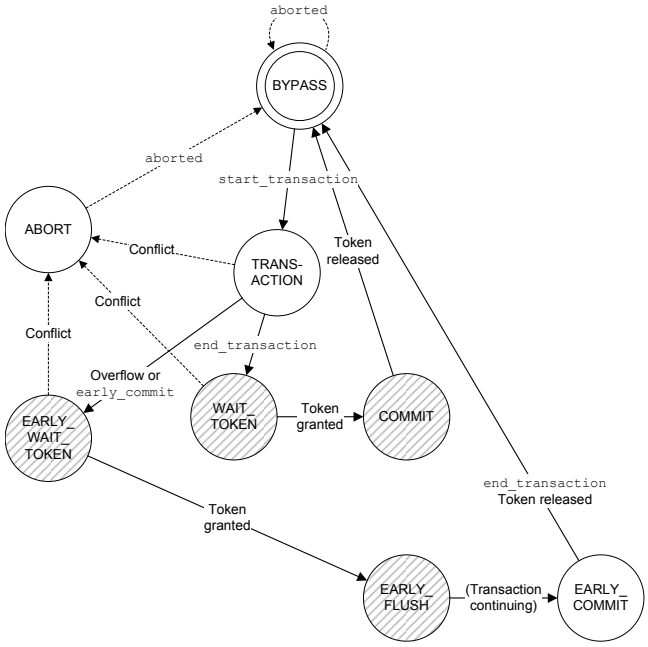

Figure 3. States of the RTTM module

Figure 2 shows a CMP system with the core local write buffers and the read tags. The shared memory includes the memory access arbiter. The commit token arbitration is also centralized.

A. A Java Chip-Multiprocessor

The RTTM is integrated in a CMP version of the Java processor JOP. JOP is an implementation of the Java virtual machine in hardware. The unique feature of JOP is its time-predictable execution of Java bytecodes. Therefore, it is an easy target for worst-case execution time analysis [16].

Pitter has designed a CMP version of JOP [10]. To keep this CMP system time-predictable, the access to the shared main memory is controlled by a TDMA based arbiter. The static schedule of the TDMA arbiter has been integrated into the low-level timing model of JOP in the WCET analysis tool WCA [16]. Therefore, WCET analysis is even possible for a JOP based CMP system.

B. RTTM Module Logic

The possible states of the RTTM module are shown in Figure 3. When no transaction is active, in state BYPASS, all memory accesses bypass the RTTM module. A transaction can be aborted from three different states: during normal transaction processing (TRANSACTION); when waiting for the commit token (WAIT_TOKEN); or when waiting on the commit token on a buffer overview (EARLY_WAIT_TOKEN).

C. Transaction Buffer

For the transaction buffer a simplified FIFO replacement strategy is feasible. Just a single fill counter decides on the

```
-- a plain priority encoder
l <= (others => '0');
for i in 0 to lines-1 loop
        if h(i)='1' then
                l <= to_unsigned(i, way_bits);
                exit;
        end if;
end loop;

-- encoder without priority
l <= (others => '0');
for i in 0 to way_bits-1 loop
        for j in 0 to lines-1 loop
                n := to_unsigned(j, way_bits);
                if n(i)='1' and h(j)='1' then
                        l(i) <= '1';
                end if;
        end loop;
end loop;
```

Figure 4. VHDL code for a standard priority based encoder and an optimized version without priority.

# cores	Total (LC)	RTTM (LC)	Memory	Fmax
1	3383	-	8.1 KB	107.3 MHz
2	10011	3199	16.7 KB	102.7 MHz
4	19877	6390	33.3 KB	103.7 MHz
8	39610	12806	66.7 KB	94.9 MHz

Table I
IMPLEMENTATION RESULTS FOR A CMP SYSTEM WITH 32 WORD TRANSACTION BUFFERS

# ways	Total (LC)	RTTM (LC)	Memory	Fmax
16	18201	4727	32.9 KB	102.8 MHz
32	19877	6390	33.3 KB	103.7 MHz
64	23118	9669	34.2 KB	100.4 MHz
128	29534	16086	35.8 KB	92.8 MHz
256	42421	28940	39.1 KB	84.3 MHz
512	66283	53108	45.8 KB	75.3 MHz

Table II
IMPLEMENTATION RESULTS FOR A 4 CORE SYSTEM WITH FULLY ASSOCIATIVE TRANSACTION BUFFERS OF DIFFERENT SIZES

next element that will be used. After a successful commit or on an abort, the transaction buffer is invalidated by resetting the fill counter. If the fill counter reaches the maximum value, the overflow mechanism with the early commit is triggered.

The critical part of the tag memory is the hit detection. All tag memories have to be compared with the address and the correct line be chosen on a hit. It is easy to unintended code a priority encoder for the line index generation. This priority selection generates additional logic in the critical path. Carefully written VHDL code implements a simpler encoder.

Figure 4 shows the VHDL code for a priority encoder and an encoder without priorities. Signal h(i) is 1 if the tag memory for line i matches the address. At any time only one or none of the hit signals is 1. Signal l gives the line index on a hit. If there is no hit, the line index is simply ignored.

D. Memory Access Types

Not all memory accesses need to be written to the transaction buffer or recorded in the read set. As the stack and local variables are thread local in Java, those memory accesses are not considered part of a transaction. Furthermore, access to constant data, such as instruction loads and loads from the Java constant pool, are also excluded from the read set. Only access to object fields and static data forms the global state and is considered part of the transaction. This reduction helps to keep the buffer demand lower than with traditional cache based HTMs.

E. Results

To evaluate the resource requirements of RTTM and the influence of the tag memory on the operation frequency we have synthesized several versions of the JOP CMP system. The target FPGA is the Altera Cyclone-II FPGA EP2C70-6. The design was constrained with a maximum clock frequency of 100 MHz.

Table I shows variations in the number of processor cores with a transaction buffer of 32 words. The first column gives the number of cores, the second the resource of the whole CMP system including the RTTM modules. The third column shows the resource consumption of the RTTM modules (included in the second column). Column 4 shows the on-chip memory consumption and the last column gives the maximum operation frequency. A single core version of JOP without RTTM serves as reference design.

JOP with some hardware support for object-oriented operations consumes 3400 logic cells (LC). Support of a 32 word transaction buffer for a dual core CMP version consumes almost as many LC as a single processor core. As the tag memory is fully associative, it is implemented in discrete registers. For 4 and 8 cores the size increases roughly linear. The RTTM with 32 words has no influence on the maximum operation frequency. The critical path is in the memory arbiter and depends on the number of cores.

Table II presents resource consumption and maximum frequency for a 4 core CMP with variations of the buffer size. The resource consumption of the RTTM buffer increases less than linear, because part of the design has only logarithmic complexity. Starting at a buffer size of 128 words the RTTM dominates the resource consumption of the whole CMP system. From that point on the RTTM also limits the maximum operation frequency. However, it is surprising that a 512 way associativity can be implemented at 75 MHz within the Cyclone-II FPGA. If this high associativity is needed the hit detection can be pipelined, resulting in an additional cycle per memory access.

In summary, our implementation shows that RTTM is fea-

sible at a moderate resource consumption. High associativity, without degrading the clock frequency, can be achieved with a non-priority encoding of the hit detection.

VI. Conclusion

This paper presents the design and implementation details of a new synchronization paradigm for hard real-time systems. Real-time transactional memory provides simple atomic sections to protect shared data in concurrent programs. To keep the transactions analyzable, the transaction buffer has to avoid false conflict detections. The resulting design is a fully associative buffer, with FIFO replacement, and single word lines. The evaluation in a Cyclone-II FPGA shows that an associativity up to 64-way is feasible. 64 words restrict the practical transaction size, but greatly simplify implementation of non-blocking algorithms and data structures.

As future work we plan to adapt some real-world benchmarks from lock based synchronization to transactions. We will compare the performance and the code complexity. Furthermore, we want to compare wait-free algorithms with versions that can explore the power of transactions.

The RTTM implementation and the Java processor JOP are open-source under the GNU GPL. The sources are available from the git repository git://www.soc.tuwien.ac.at/jop.git.

Acknowledgement

This research has received partial funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

References

[1] M. Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universitat Berlin, April 2001.

[2] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. K. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, pages 1–13, Boston, MA, USA, October 2004. ACM.

[3] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[4] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the OOPSLA '03 conference*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[5] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on*, pages 289–300, 1993.

[6] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 2006 OOPSLA Conference*, pages 253–262, New York, NY, USA, 2006. ACM Press.

[7] T. Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, 1986. ACM Press.

[8] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 62–71, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[9] C. Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 115–122, Santa Clara, USA, September 2008. ACM Press.

[10] C. Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[11] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[12] M. Schoeberl. Java technology in an FPGA. In *Proceedings of the International Conference on Field-Programmable Logic and its Applications (FPL 2004)*, pages 917–921, Antwerp, Belgium, August 2004. Springer.

[13] M. Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[14] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[15] M. Schoeberl, F. Brandner, and J. Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC 2010)*, pages 326–333, Sierre, Switzerland, March 2010. ACM Press.

[16] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[17] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[18] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A practical FPGA-based framework for novel CMP research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, New York, NY, USA, 2007. ACM.